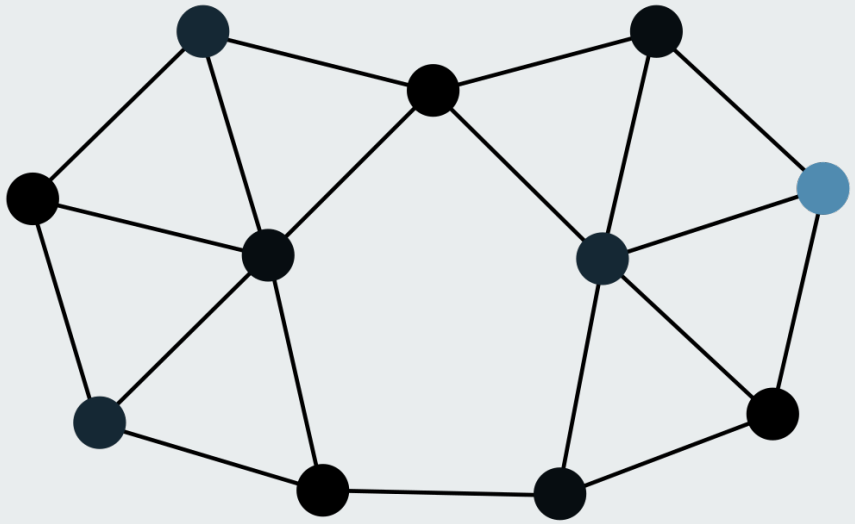
Distributed Systems

Processes



Processes

...are instances of programs running on operating systems.

In a Distributed system, management and scheduling is very important (particularly when migrating processes)

Threads

Improve performance of **clients** and **servers**.

Communication and **computation** can overlap for better performance.

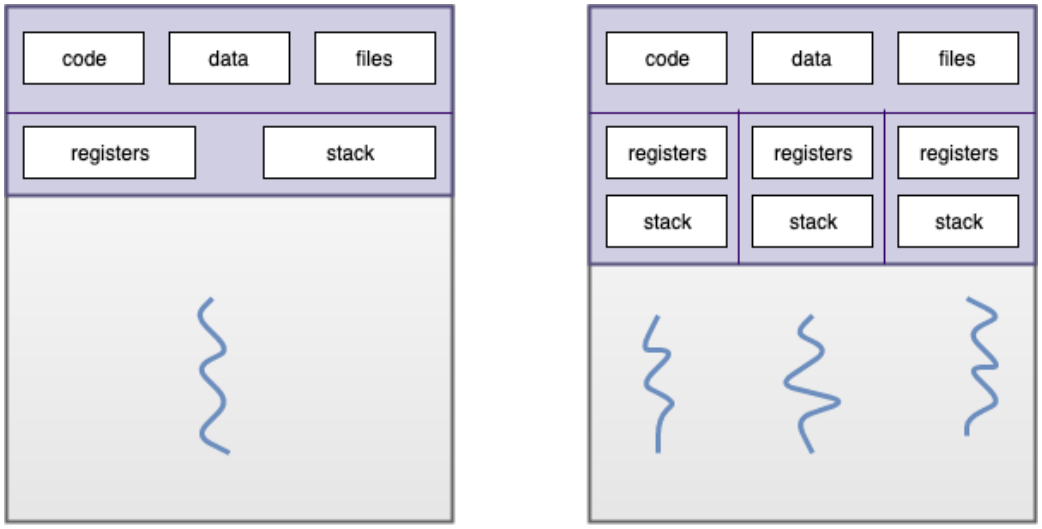
Simplify the structure of applications.

By offloading communication to other threads, can use blocking calls (synchronous calls) while overlapping computation.

Thread vs Process

Thread: Shared memory space (a lightweight process). Used for small tasks.

Process: Own memory space. Execution of applications.



Single threaded process

Multi-threaded process

Multithreaded Clients

Round trips can be hundreds of milliseconds (especially in WANs), so blocking calls cause a lot of idle time.

- Overlap is performed in web browsers by using multi-threads… multiple TCP connections are threaded for retrieving resources.
- Single thread for rendering & user input.
- Speeds up UX

SERVERS

Multithreaded Servers

Overlapping communication with computation:

- More efficient use of resources (Higher throughput)
- Simplify development

There are many different ways in which a server can be implemented

We will discuss three: **threaded**, **single thread** and **finite state machine**

Multithreaded Servers

Have a single thread listening for requests on a port (dispatcher)

Whenever a request is made, the dispatcher passes the request to an idle worker

While it is possible that workers are blocked (performing disk I/O or communication) others can overlap

Leads to higher throughput, better performance, and simplifies development

Single Threaded Servers

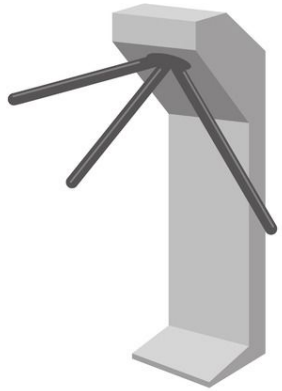
Here each request is served one at a time.

Blocking calls cause the server to sit idle.

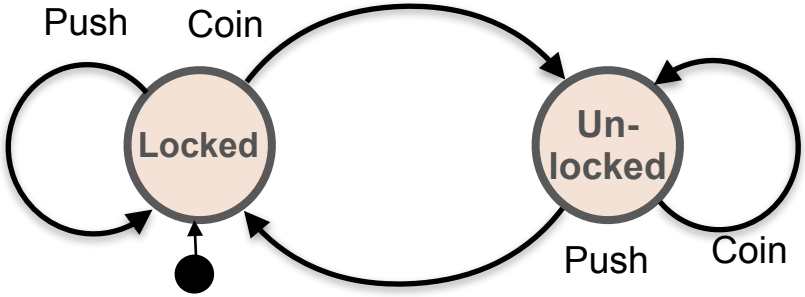
Prevents full use of the server & reduces performance.

Will always underutilise resources.

Finite-State Machine



Model of computation.



A single thread is used. Each request and its state recorded in a table.

When disk I/O or communication is made it will use a non blocking call and will move to the next request

When the request returns it will finish the original request.

Enables parallelism and overlapping, but increases code complexity.

Servers

A server exists solely to provide a service to one or more clients

Most servers are structured in the same way:
Wait for a request from a client, process that request, repeat ad infinitum

Iterative Vs Concurrent Servers

- **Iterative** handle requests and return a response itself. Handles each request in turn.
- **Concurrent** pass requests to an idle thread or process and immediately wait for the next request

Super Servers

Implementing several servers may be wasteful.
Alternative? Super server.

- One process listens on all required ports
- When a request is received, it forks a process of the relevant service for that request.
- Once the request finishes, the forked process dies.

Example: UNIX/Linux inetd service

Server Interruption

A client should be able to interrupt a server if they wish to cancel a request.

- **Server timeout**: the client abruptly exits and cause the server to timeout on the connection. After the timeout the server will tear down the dead connection.
- **Out-of-band data:** data that's processed before any other data is received by the server. *Could be sent on the same or a different port as the data transfer.*

Stateless/Stateful Servers

Stateless servers keep no information about the state of clients

- Can change own state without having to inform the clients
- Example: HTTP server that only responds to requests

Stateful servers maintain persistent information about clients

- This information must be explicitly deleted by the server
- If a server crashes, it must recover client states
- Two states: session state vs permanent state

Session State vs Permanent State

Session state is associated with a series of operations by a single user and should be maintained for a limited time.

- If this information is lost no harm is done once the client can reissue the same request

Permanent state is information contained in databases such as user information.

MIGRATION

Process Migration

Important for **failure tolerance** and **scalability**.

Moving processes to less loaded node increases performance.

Enables dynamic configuration of the environment

Less disruption of the application/service

Mobility

Weak: only the process is moved.

- The minimum requirement for code migration.
- Moved process has a predefined position to start from

Strong: both process and state are transferred

- Process restarts in the exact same state it was paused at

Resource Migration

What makes code migration particularly difficult is the migration of local resources.

Resource Types:

Unattached: easy to move between nodes (static data files) 

Fastened: difficult to move e.g. database or web site 

Fixed: cannot be moved, bound to a specific node e.g. hardware 

Migration in Heterogeneous Systems

Heterogeneous systems require code to be executed regardless of architecture

Such as using **machine independent intermediate instructions** (Java/Pascal)

Virtualisation

This is where virtualisation becomes useful as the entire computing environment can move with the application

Enables **processes** or **environments** to move between nodes.

Applications can relocate regardless of the underlying hardware

Applications become very portable

Helps with failure tolerance

If you don't know much about this play with:

VirtualBox

Virtual Machines

Achieved by providing an **abstract instruction set** to applications.

Abstract instructions translated to **machine instructions**.